操作系统课程设计

项目试验报告

项目五(Final)

陈士凯
5040309017

目录

实验：USB MASS STORAGE 驱动实现

实验目标：

通过对 Linux 内核中自带的 usb 大容量存储设备驱动:usb-storage 的分析，制作支持特定 u 盘的驱动模块。要求具有正常 usb 大容量存储设备驱动应有的功能。

本次试验中使用的 u 盘是本人的 SAMSUNG D828 手机，具有移动磁盘功能。实验实现了对该设备 u 盘功能的驱动。

原理：[1]

USB MASS STORAGE 协议分析

本次试验首先参考了 *Universal Serial Bus, Mass Storage Class Specification Overview* 白皮书，其中对 USB Mass Storage 的说明如下：

其中，对于 USB Mass Storage 的数据通讯协议分为下面几类：

Sub Class	协议名	说明
0x01	Reduced Block Commands(RBC)	通常为 Flash Rom 介质的存储设备使用
0x02	8020i, MMC-2(ATAPI)	通常为 CD/DVD 设备使用
0x03	QIC-157	常用于磁带机设备
0x04	UFI	常用于软磁盘设备(FDD)
0x05	8070i	常用于软磁盘设备或者其他设备
0x06	SCSI 协议	
0x07-0xFF	保留	

摘录至 *Universal Serial Bus, Mass Storage Class Specification Overview, P6*

Sub Class 可以在 usb 设备连入系统后获取。不同的通讯协议决定了 usb 驱动要用不同的命令和数据包格式和 u 盘通讯。

同时，白皮书还规定了 u 盘设备的通讯方式，见下表：

接口协议号	协议名
0x00	Control/Bulk/Interrupt 协议(CBI) 带有 command completion 中断
0x01	Control/Bulk/Interrupt 协议(CBI) 不带有 command completion 中断
0x50	Bulk-Only 协议
0x02-0x4F	保留
0x51-0xFF	保留

摘录至 *Universal Serial Bus, Mass Storage Class Specification Overview, P7*

[1] 本次试验所用内核版本为 2.6.20.3

接口协议号即 InterfaceProtocol 字段，在 usb 设备插入后可以从总线上获取。

在实际 usb mass storage 设备工作时，设备驱动首先要做的是获得该 u 盘的通讯协议和通讯方式，然后按照需要产生与目标设备兼容的控制命令，并将该控制命令打包通过设备的通讯传输方式发送至设备，完成一次对设备的读写操作。

对于上述 6 中数据通讯协议，均有很详细的说明文档，经过分析，他们的命令有很大的相似之处，这也是通讯 usb 驱动能够实现的原因之一。

自带驱动 usb-storage 的简要原理分析

linux 内核中自带的 usb mass storage 驱动位于内核源代码目录/drivers/usb/storage/下，下表为该目录下各文件的功能说明：

文件	功能说明
usb.c/.h	Usb-storage 的核心文件，是整个驱动的框架代码
transport.c/.h	实现了对于不同通讯方式的支持函数
scsiglue.c/.h	Scsi 设备的模拟函数
protocol.c/.h	实现了对于几种通讯协议的 SCSI 命令翻译函数
initializers.c/.h	对于某些设备的专用初始化函数
unusual_devs.h	对于非常规设备 ProductID 和 VendorID 的支持
shuttle_usbat.c/.h	支持 SCM Microsystems 设备的驱动
sddr55.c/.h	SanDisk SDDR-55 SmartMedia reader 的驱动
sddr09.c/.h	anDisk SDDR-09 SmartMedia reader 的驱动
onetouch.c/.h	Maxtor OneTouch USB hard drive 驱动支持
libusual.c	对于常规设备的 ProductID 和 VendorID 的支持
karma.c/.h	Rio Karma 设备驱动
jumpshot.c/.h	Lexar "Jumpshot" Compact Flash reader 驱动
isd200.c/.h	ISD200 专属通讯协议支持
freecom.c/.h	Freecom USB/IDE 转化器支持
dpcm.c/.h	DPCM-USB CompactFlash/SmartMedia reader 设备支持
debug.c/.h	用于调试的工具函数
datafab.c/.h	Datafab USB Compact Flash reader 驱动支持
alauda.c/.h	Alauda-based card readers 驱动支持

通过上述文件作用分析后，可以看出，对于常规的 usb 设备，大部分代码文件都是多余的，核心文件为上表的前 4 项。这就为进一步分析驱动已经对其进行简化和仿制提供了可能。

通过对核心代码的分析，本驱动的工作机理是将自身模拟为标准的 SCSI 设备，并向 scsi 管理器注册，这样对于上层系统而言，只需操作标准的 SCSI 设备即可。这样可以简化具体的文件读写功能。同时驱动接受到的 SCSI 命令转化为对应 u 盘设备的通讯协议，并用对应设备的通讯方式进行发送，并将结果回馈到 SCSI 管理器。

下图展示了整个驱动的工作布局：

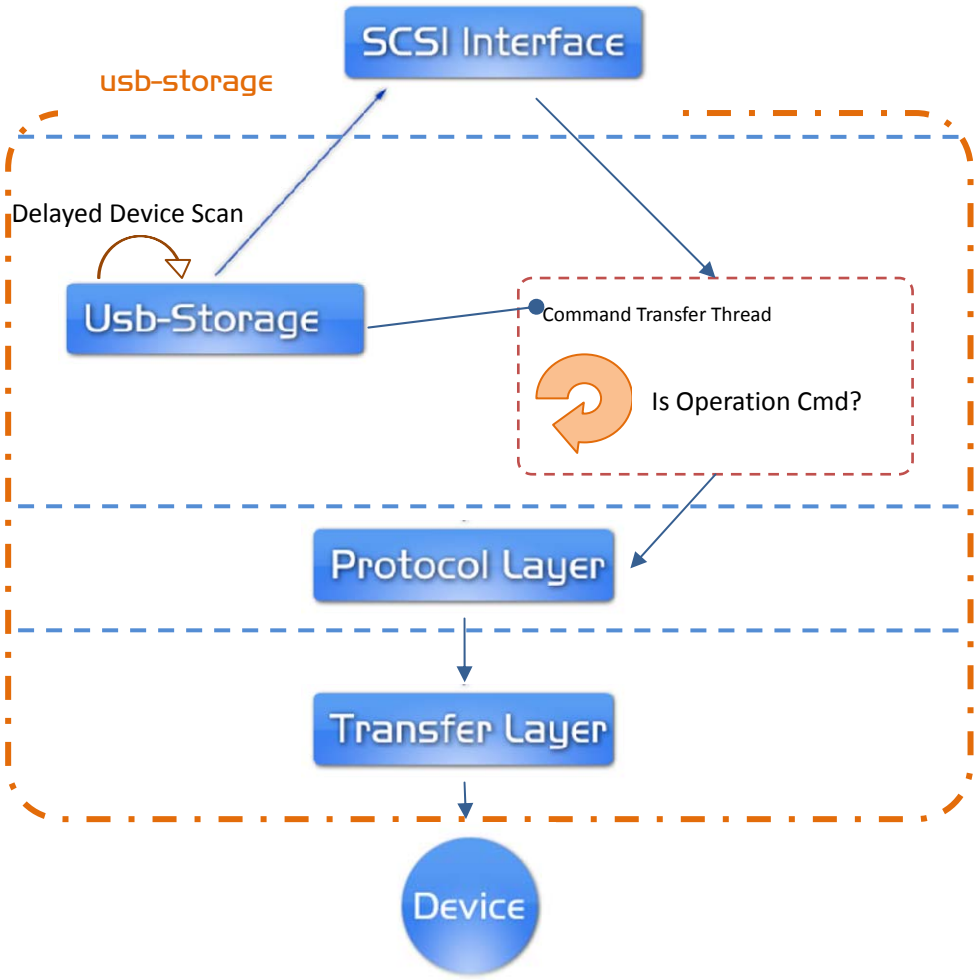

图:usb-storage 驱动工作机制

图中 Transfer layer 和 protocol layer 分别对应了 transfer.c 和 protocol.c 文件，Command Transfer Thread 是 usb.c 在探测到有新设备接入后加载的线程，他将不断轮询 SCSI 发来的消息命令，并负责将这些命令通过 protocol layer 提供的函数翻译并发送到 u 盘上，完成对 u 盘的读写操作。

Delayed Device Scan 是为了防止用户在设备插入后马上拔除，造成驱动在后续通讯中造成混乱。实现方式是创建一个专门线程。

同时，usb.c 在设备插入是会通过 scsiglue.c 提供的 SCSI 接口函数向 SCSI 管理器注册自身。

- 对 usb-storage 的简化

经过上面对 usb-storage 的分析，可以将该驱动代码作如下方面的简化，方便后续的分析和驱动编写。

- 删除其中对罕见设备的支持
- 删除 Delayed Device Scan 机制
- 删除对于除了本次试验所用 u 盘设备外其他的通讯协议和通讯方式的处理函数。
- 对于仅调用一次的函数作 inline 处理

其中，每一项操作都会影响到原先的函数依赖关系，因而需要在删除代码或者文件前弄

清楚其中的依赖关系。

D828 的设备通讯协议分析

通过使用软件 usbview 可以得知如下数据：

项目	值
ProductID	0x665c
VendorID	0x04e8
Protocol	0x50
Subclass	0x02

可以得知，D828 的 u 盘功能使用了 ATAPI 通讯协议，且使用 Bulk-only 方式进行数据传输。因此，在编写驱动时候可以仅处理以上情况。

实现过程：

将 usb-storage 从内核中移除

在配制内核时候，将 usb-storage 从编译选项中移除。否则在插入 u 盘是，usb-storage 将自动被挂载，导致无法执行本次试验编写的驱动。

对 usb-storage 的简化

按照原理中提到的步骤，首先是删除其对罕见驱动和专属驱动的支持。最终将保留下上面表格中前 4 项文件。

同时处理 usb.c 对这些代码的引用，主要是在如下代码段中：

```
126 #ifndef CONFIG_USB_LIBUSUAL
127
128 #define UNUSUAL_DEV(id_vendor, id_product, bcdDeviceMin, bcdDeviceMax, \
129                     vendorName, productName,useProtocol, useTransport, \
130                     initFunction, flags) \
//...//
181 #       undef USUAL_DEV
182
183         /* Terminating entry */
184         { NULL }
185 };
186
187
188 #ifdef CONFIG_PM        /* Minimal support for suspend and resume */
```

代码：原始 usb.c 中代码片断

在这个区间中的代码将设置对不同设备，包括罕见设备 ProductID 和 VendorID 的支持。

可以改为仅对 D828 设备的支持：

```
//SAMSUNG MOBILE USB INFO
#define USB_MASS_VENDOR_ID      0x04e8
#define USB_MASS_PRODUCT_ID     0x665c

//REG IT
static struct usb_device_id storage_usb_ids [] = {
      { USB_DEVICE(USB_MASS_VENDOR_ID, USB_MASS_PRODUCT_ID) },
      { }                       /* Terminating entry */
};

MODULE_DEVICE_TABLE (usb, storage_usb_ids);
```

代码：替换过 usb.c 后的代码

当然，在后续的代码中，仍需要进行修改，比如 us_unusual_dev_list[]在后续的引用也需要删除。

在经过第一个简化操作后，便可以对 D828 协议以外的处理函数进行删除。在 protocol.c 中存在如下几个函数：

```
void usb_stor_qic157_command(struct scsi_cmnd *srb, struct us_data *us);
void usb_stor_ATAPI_command(struct scsi_cmnd *srb, struct us_data *us);
void usb_stor_ufi_command(struct scsi_cmnd *srb, struct us_data *us);
void usb_stor_transparent_scsi_command(struct scsi_cmnd *srb, struct us_data *us);
```

代码：protocol.c 中提供的函数原形

这正是针对白皮书中规定的 4 种不同的通讯协议的处理函数，而他们是通过 usb.c 中如下代码片断和当前设备进行联系的：

```
671 /* Get the protocol settings */
672 static int get_protocol(struct us_data *us)
673 {
674         switch (us->subclass) {
675         case US_SC_RBC:
676                 us->protocol_name = "Reduced Block Commands (RBC)";
677                 us->proto_handler = usb_stor_transparent_scsi_command;
678                 break;
679
680         case US_SC_8020:
681                 us->protocol_name = "8020i";
682                 us->proto_handler = usb_stor_ATAPI_command;
683                 us->max_lun = 0;
684                 break;
```

代码：usb.c 中 get_protocol 函数部分

可以看到每个上述函数按照当前设备的 subclass 信息被赋值到 us->proto_hander 指针中。

因而，可以只保留其中针对 D828 的函数，即 usb_stor_ATAPI_command，同时删除 get_protocol 函数。

基于同样的原理，可以对 transfer.c 和 get_transport 作相同的处理。

接下来是对 usb.c 中的 delayed device scan 机制进行删除。因为本次试验不需要适应现实中各类特殊情况。

在 usb.c 中 usb_stor_scan_thread 完成了对设备的延迟探索功能，其中代码段：

```
926                     /* For bulk-only devices, determine the max LUN value */
927                     if (us->protocol == US_PR_BULK &&
928                                             !(us->flags & US_FL_SINGLE_LUN)) {
929                             mutex_lock(&us->dev_mutex);
930                             us->max_lun = usb_stor_Bulk_max_lun(us);
931                             mutex_unlock(&us->dev_mutex);
932                     }
933                     scsi_scan_host(us_to_host(us));
934                     printk(KERN_DEBUG "usb-storage: device scan complete\n");
//...//
939             scsi_host_put(us_to_host(us));
940             complete_and_exit(&threads_gone, 0);
```

代码：usb_stor_scan_thread 函数片断

上述代码完成了对设备的 scan 操作，而其他部分则与实际工作无关，因而可以将上述代码复制到 storage_probe 函数中对应位置，并删除函数 usb_stor_scan_thread。

- 修改 usb-storage 的 Makefile，使得其能够单独编译并工作[2]

首先将 storage 目录另外复制到其他目录，其中的 Kbuild 已经对本次试验无用，可以删除。

分析其中的 Makefile，将其改写成如下代码：

```
EXTRA_CFLAGS      := -Idrivers/scsi
#obj-$(CONFIG_USB_STORAGE)    += usb-storage.o
obj-m      := csk-usb-storage.o


csk-usb-storage-objs :=   scsiglue.o protocol.o transport.o usb.o \
                initializers.o

all:
        make -C /usr/src/linux-headers-2.6.20.3csk1 M=$(shell pwd) modules
clean:
        make -C /usr/src/linux-headers-2.6.20.3csk1 M=$(shell pwd) clean
```

[2] 经过裁剪的代码见本次附带的 simplified version 目录

接下来通过 make 进行编译，如果一切顺利，在 insmod 本驱动后，就能完成 u 盘的操作。

- **制作专用驱动**[3]

在经过对系统自带代码的裁剪操作后，可以通过利用其中设备通讯等工具函数，以及仿造其中的框架重新制作 D828 的专属驱动。

a) 该操作按照下面的流程进行：
b) 放造 usb.c 编写框架代码
c) 将框架代码需要调用的工具函数和结构定义复制进本驱动代码
d) 在上次复制的函数中找到需要引用但尚未复制的函数，将他们添加近来
e) 查找仅引用过一次的函数，作 inline 处理
f) 其他的一些优化

经过上述步骤以后，只要编译通过，该驱动便能顺利运行了。

试验结果：

通过 insmod 将本次试验完成的驱动加入内核后，在插入 D828 设备就会自动挂在 u 盘文件系统。同时能进行所有 u 盘常规操作。

至此，全部试验内容完成。

---

[3] 重新制作的驱动见 reconstructed version 目录